IIF
INSTITUT FÜR
INFORMATIK
FREIBURG

Vorlesung Informatik 2
Algorithmen und Datenstrukturen

(01 – Einleitung)

Prof. Dr. Susanne Albers

Organisatorisches

Vorlesung: Mo 14.00 – 16.00 Uhr, HS 00-026, Geb.101
Do 11.00 – 13.00 Uhr, HS 00-026, Geb.101

Übungen: Anmeldung bis
- Mo 9-11, SR 03-026, Geb. 051
- Mo 9-11 SR 00-034, Geb. 051
- Mo 9-11 SR 01-018, Geb. 101
- Mo 16-18 SR 03-036, Geb. 051
- Mo 16-18 SR 01-018, Geb. 051
- Do 16-18, SR 01-018, Geb. 101
- Do 16-18, SR 03-026, Geb. 051

http://www.informatik.uni-freiburg.de/~ipr/
→ Teaching → Informatik II

Organistorisches

Klausur: 18. Oktober 2006

Teilnahmevoraussetzung:

- 50% der Übungen bearbeitet
- 1-mal in den Übungen vorgerechnet

Bonus bei bestandener Klausur:

Verbesserung um

- 1/3 Notenstufe, wenn 50% der Übungspunkte erzielt
- 2/3 Notenstufe, wenn 80% der Übungspunkte erzielt

Literatur

Ottmann, Widmayer : Algorithmen und Datenstrukturen, Spektrum Akademischer Verlag, Heidelberg, Berlin; ISBN: 3-8274-1029-0, 4. Auflage, 2002

Saake, Sattler : Algorithmen und Datenstrukturen: eine Einführung mit Java, dpunkt-Verlag, Heidelberg, 2002; ISBN: 3-89864-122-8

Cormen, Leiserson, Rivest, Stein : Introduction to Algorithms, Second Edition, MIT-Press and McGraw Hill, 2002

Kleinberg, Tardos : Algorithm Design, Pearson/Addison-Wesley; ISBN: 0-321-29535-8, First Edition, 2005

Goodrich, Tamassia : Data Structures and Algorithms in Java, John Wiley & Sons; ISBN: 0-471-38367, Second Edition, 2001

Zahlreiche weitere Bücher von :
D. Knuth, S.Baase / Van Gelder, R. Sedgewick, Nievergelt / Hinrichs, Güting / Dieker, Heun, A. Drozdeck, Th. Standisch, Kruse, Wood, u.v.a.

Inhaltsübersicht

1. Einleitung, Grundlagen
2. Algorithmenentwurfstechniken
3. Elementare Datenstrukturen
4. Sortieren, Suchen, Auswahl
5. Wörterbücher, Bäume und Hash-Verfahren
6. Graphenalgorithmen

Lernziele

Algorithmen für wichtige Probleme :

Sortieren, Suchen, Wörterbuch-Problem, Berechnung kürzester Pfade, . . .

Datenstrukturen :

Listen, Stapel, Schlangen, Bäume, Hash-Tabellen, . . .

Problemlösetechniken :

Divide-and-Conquer, Greedy, vollständige Aufzählung, Backtracking, . . .

Ziele:

Entwicklung effizienter Algorithmen für Instanzen von Problemen aus einem gegebenen Bereich

Fähigkeit zur Beurteilung von Algorithmen aufgrund präziser Kriterien (Korrektheit, Effizienz)

Beschreibung und Analyse von Algorithmen

- Sprache zur Formulierung von Algorithmen

  Natürliche Sprache, Pseudocode, Flussdiagramme, Programmiersprache (Java, C, ...)

- Mathematisches Instrumentarium zur Messung der Komplexität (Zeit- und Platzbedarf):

  Groß-O-Kalkül (Landausche Symbole)

Pseudocode

- abstrakte Beschreibung eines Algorithmus
- strukturierter als Beschreibung mit normalem Sprachvokabular
- weniger detailliert als ein Programm
- bevorzugte Notation zur Beschreibung eines Algorithmus
- versteckt Programmentwurfsprobleme

Beispiel : Finden des größten Elements in einem *array*

Algorithmus *arrayMax(A,n)*
Input array *A* mit *n* Integern
Output größtes Element von *A*

```
currentMax ⟵ A[0]
for i ⟵ 1 to n - 1 do
    if A[i] > currentMax then
        currentMax ⟵ A[i]
return currentMax
```

Pseudocode Details

- Kontrollfluss
  - if … then … [else …]
  - while … do …
  - repeat … until …
  - for … do …
  - Einrücken ersetzt Klammern
- Deklaration von Methoden

  Algorithm method(arg[, arg…])

  Input …

  Output …
- Methodenaufruf

  var.method(arg[,arg…])
- Rückgabewert

  return Ausdruck
- Ausdrücke

  ← Zuweisung

  (wie = in Java)

  = Gleichheitstest

  (wie == in Java)

  n^2 Superscripts und andere mathematische Formatierungen sind erlaubt

Formale Eigenschaften von Algorithmen

- Korrektheit
- Effizienz

Fragen :

Wie beweist man die Korrektheit ?

→ Programmverifikation

→ Testen

Wie misst man die Effizienz von Algorithmen ?

→ Implementation und Test für „repräsentative“ Beispiele

→ Platz- und Zeitbedarf auf Real RAM

→ Bestimmung signifikanter Parameter

Korrektheit

- partielle : Wenn der Algorithmus hält, dann liefert er das gewünschte Resultat
- totale : Algorithmus hält und liefert das gewünschte Ergebnis

Vorbedingung (Eingabe-Bedingung) : Spezifiziert den Zustand vor Ausführung eines Algorithmus

Nachbedingung (Ausgabe-Bedingung) : Spezifiziert den Zustand nach Ausführung des Algorithmus

{P} S {Q}

Beispiel :

{ ? } x = y + 23 { x > 0 }

Hoare Kalkül

Grundbausteine imperativer Sprachen

Sprachkonstrukte spiegeln die Von Neumann Rechnerarchitektur wieder.

Variablen sind symbolische Namen für Speicherplätze.

Anweisungen:

Zuweisung: x = t

Komposition: S_1 ; S_2

Selektion: **if** B **then** S_1 **else** S_2

Iteration: **while** B **do** S

Hoare Kalkül: Dient zum Nachweis der partiellen Korrektheit von Programmen, d.h. von Aussagen der Form {P} S {Q}

Beweisregeln

Wenn aus wahren Aussagen $A_1, \ldots, A_n$ folgt, dass auch die Aussage A wahr ist, notiert man das in der Form:

$$\frac{A_1, \ldots, A_n}{A}$$

Die zu verifizierenden Aussagen sind die Hoare-Tripel der Form

{P} S {Q}.

Zuweisungsaxiom

$$\{P[x/t]\}\ \ x = t\ \ \{P\}$$

P[x/t] bedeutet, dass in der Aussage P jedes Vorkommen von x durch den Term t ersetzt wird.

Beispiele:

$$\{x+1 = 43\}\ \ y = x+1 \quad \{y = 43\}$$

$$\{y = 43\} \quad z = y \quad \{z = 43\}$$

Komposition und Selektion

Komposition

$$\frac{\{P\}\ S_1\ \{Q\},\ \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1;\ S_2\ \{R\}}$$

Selektion

$$\frac{\{P \text{ und } B\}\ S_1\ \{R\}\ ,\quad \{P \text{ und nicht } B\}\ S_2\ \{R\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{R\}}$$

Beispiel zur Komposition

Komposition

$$\frac{\{P\}\ S_1\ \{Q\},\ \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1;\ S_2\ \{R\}}$$

{x+1 = 43}

y = x+1

{y = 43}

z = y

{z = 43}

Iteration

P impliziert I, {I und B} S {I}, (I und nicht B) impliziert Q

{P} **while** B **do** S {Q}

Die in dieser Regel auftretende Aussage I heißt Schleifeninvariante.

Das Finden geeigneter Schleifeninvarianten ist i.a. algorithmisch unlösbar!
Daher empfiehlt es sich, Schleifeninvarianten als Kommentare (Assertions) an den entsprechenden Stellen in Programme einzufügen

Abschwächung und Verstärkung

Abschwächung

$$\frac{\{P\} \Rightarrow \{P'\},\ \{P'\}\ S\ \{Q\}}{\{P\}\ S\ \{Q\}}$$

Verstärkung

$$\frac{\{P\}\ S\ \{Q'\},\ \ \{Q'\} \Rightarrow \{Q\}}{\{P\}\ S\ \{Q\}}$$

Beispiel eines (formalen) Korrektheitsbeweises

Algorithmus *Mult(x,y)*

Eingabe : Ein Paar x, y von natürlichen Zahlen

Ausgabe : Das Produkt von x und y

Methode :

```
z ⟵ 0 ;
while (y>0) do {
    if (y ist gerade)
        then {y ⟵ y/2; x ⟵ x+x}
    else /* y ist ungerade */
            {y ⟵ y-1; z ⟵ z+x}
    }
return z;
```

Implementation in Java

```
class Mult {

    public static void main ( String [] args ) {
        int x = new Integer (args[0]).IntValue();
        int y = new Integer (args[1]).IntValue();
        System.out.println ("Das Produkt von " +x+ " und
                             " +y+ " ist " +mult(x,y));

    public static int mult (int x, int y) {
        int z = 0;
        while (y>0)
            if (y % 2 == 0) { y = y / 2; x = x+x ;}
            else { y = y-1; z = z+x; }

        return z;
    }
}
```

Durchführung von *Mult(x,y)* an einem Beispiel

Durchführung von *Mult(x,y)* an einem Beispiel

```
1101*101
   1 101
  00 00
 110 1
1000 001
```

x	y	z	# Iterationen

Nachweis der totalen Korrektheit

Beh. (1) Für jedes Paar `a,b` von natürlichen Zahlen gilt: `Mult(a,b)` hält nach endlich vielen Schritten .

Beh. (2) Sind `a` und `b` natürliche Zahlen, dann liefert `Mult(a,b)` den Wert `z = a * b`.

```
int z = 0;
while (y>0)
        if (y % 2 == 0) { y = y / 2; x = x+x ;}
        else { y = y-1; z = z+x; }
```

Schleifeninvariante

Scheifeninvariante I: $y \geq 0$ und $z + x\,y = a\,b$

Beh. 2.1: I gilt vor erstmaliger Ausführung der while-Schleife

Beh. 2.2: I bleibt bei einmaliger Ausführung des Rumpfs der while-Schleife richtig.

```
int z = 0;
while (y>0)
        if (y % 2 == 0) { y = y / 2; x = x+x ;}
        else { y = y-1; z = z+x; }
```

Weiteres Beispiel

{n ≥ 0}

    i = 0; k = -1; y = 0;

    while i < n do

        i = i + 1; k = k + 2; y = y + k

{y = n^2}

Schleifeninvariante:

(k = 2i -1) und (y = i^2) und (i ≤ n)

Beschreibung und Analyse von Algorithmen

Sprache zur Formulierung von Algorithmen :

natürliche Sprache (Englisch), Java, C, Assembler, Pseudocode

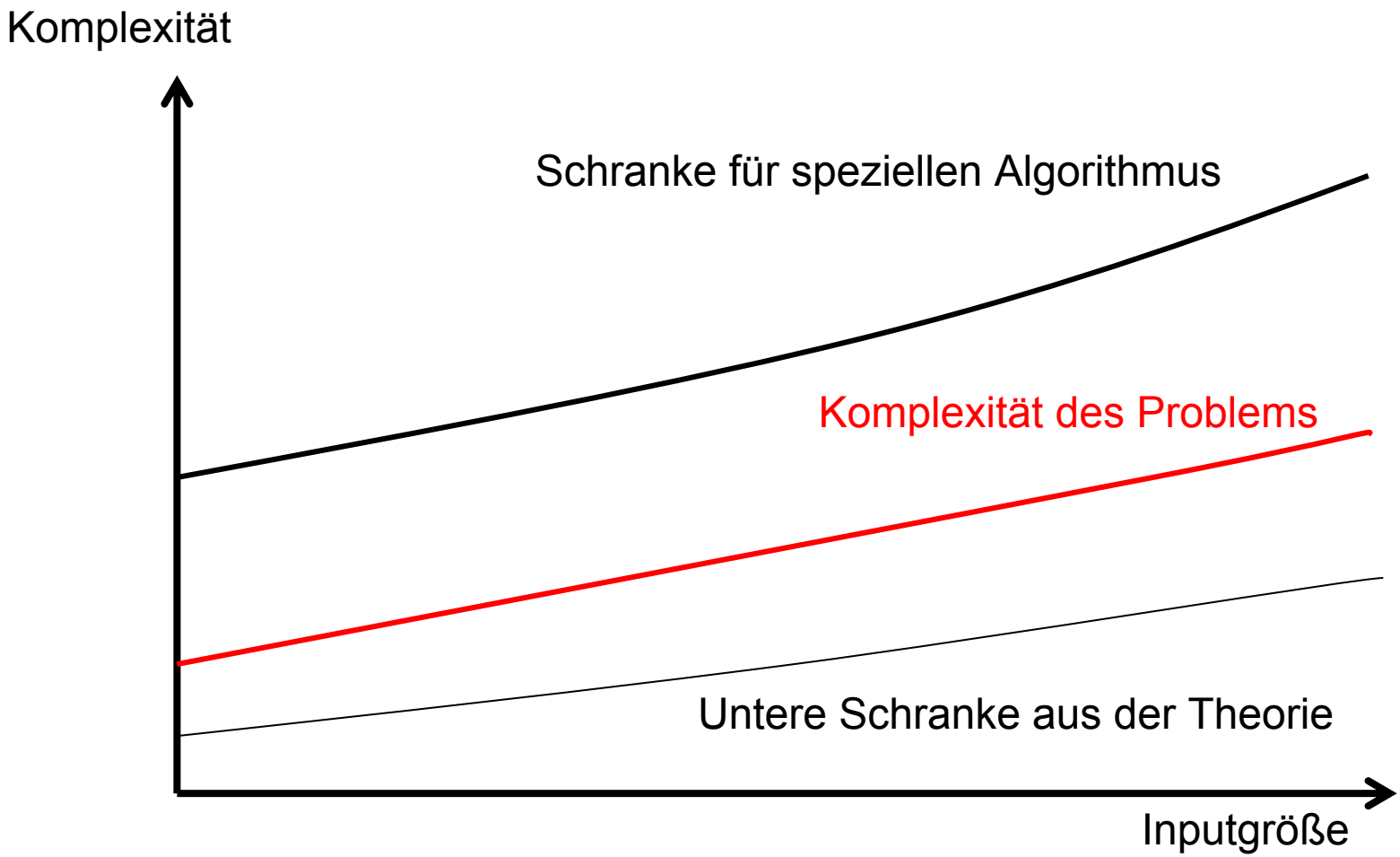
Mathematisches Instrumentarium zur Messung der Komplexität (Zeit- und Platzbedarf):

Groß-O-Kalkül (Landausche Symbole)

Effizienzanalyse

- Laufzeitkomplexität: Steht die Laufzeit im akzeptablen/vernünftigen/optimalen Verhältnis zur Größe der Aufgabe?
- Speicherplatzkomplexität: Wird primärer (sekundärer) Speicherplatz effizient genutzt?
- Theorie: Kann untere Schranken liefern, die für jeden Algorithmus gelten, der das Problem löst
  (etwa $\Omega(n \log n)$ Schritte für jedes allgemeine Sortierverfahren mit n Elementen)
- Spezieller Algorithmus: Liefert obere Schranke für die Lösung eines Problems
  (etwa $O(n^2)$ Schritte für Bubblesort mit n Elementen)
- Effiziente Algorithmen und Komplexitätstheorie: Zweige der Theoretischen Informatik zur Erforschung von oberen und unteren Schranken

Komplexitätsschranken



Laufzeitanalyse (1)

Ein Programm *P*, das für eine Problembeschreibung *x* mit Länge n=|*x*| die Lösung findet, habe Laufzeit T_P (n)

Der beste Fall (best case): Laufzeit meist leicht bestimmbar, kommt in der Praxis eher selten vor:

$$T_{P,best}(n) = \inf\{T_P(x) \mid n = |x|\}$$

Der schlechteste Fall (worst case): Liefert garantierte Schranken, Laufzeit meist leicht bestimmbar, aber meist zu pessimistisch in der Praxis:

$$T_{P,worst}(n) = \sup\{T_P(x) \mid n = |x|\}$$

Im amortisierten worst case wird der durchschnittliche Aufwand für eine schlechtestmögliche Folge von Eingaben bestimmt (technisch anspruchsvoll).

Der mittlere Fall (average case): z. B. Mittelung über alle Eingaben mit Länge n

$$T_{P,average}(n) = 1/_{(\#(x)\text{mit}|x|=n)} \sum_{|x|=n} T_P(x)$$

Messung des Leistungsverhaltens

1. Betrachte konkrete Implementierung auf konkreter Hardware. Miss Laufzeit und Platzverbrauch für repräsentative Eingaben.
2. Berechne Verbrauch an Platz und Zeit für idealisierte Referenzmaschine, Random Access Machine (RAM), Registermachine (RM), Turingmachine (TM), . . .
3. Bestimme Anzahl bestimmter (teurer) Grundoperationen, etwa
   - # Vergleiche, # Bewegungen von Daten (beim Sortieren)
   - # Multiplikationen/Divisionen (für numerische Verfahren)

Bei 2. und 3.: Beschreibe Aufwand eines Verfahrens als Funktion der Größe des Inputs. (Die Input-Größe kann verschieden gemessen werden.)

Beispiel: Taktzahl (1)

Bestimme Aufwand (Taktzahl = Anzahl der Bitwechsel) eines Von Neumann Addierwerks bei Addition einer `1` zu einer durch *n* Binärziffern gegebenen Zahl *i*.

$$0 \leq i \leq 2^n - 1$$

Die Taktzahl ist `1` plus # der 1en am Ende der Darstellung von *i*.

Bester Fall:

Die best case Rechenzeit beträgt `1` Takt

(Addiere `1` zu $\underbrace{000...0}_{n}$)

Schlechtester Fall:

Die worst case Rechenzeit beträgt `n + 1` Takte

(Addiere 1 zu $\underbrace{111...1}_{n}$)

Beispiel: Taktzahl (2)

Mittlerer Fall:

Angenommen wird die Gleichverteilung auf der Menge der Eingaben. Es gibt $2^{(n-k)}$ Eingaben, die mit $01\underbrace{\ldots}_{k-1}1$ enden und k Takte benötigen. Die Zahl $2^n - 1$ braucht $n + 1$ Takte.

Die average case Rechenzeit beträgt also

$$T_{add1}(n) = \frac{1}{2^n}\left(\sum_{1 \le k \le n} 2^{n-k} k + (n+1)\right)$$
$$= 2^{-n}(2^{n+1} - 2 - n + (n+1)) = 2 - 2^{-n}$$

Im Mittel reichen also 2 Takte, um eine Addition von 1 durchzuführen.

Nebenrechnung

$$\sum_{1 \le k \le n} 2^{n-k} k = n * 2^{n-n} + \ldots + 2 * 2^{n-2} + 1 * 2^{n-1}$$

$$= \quad 2^0 + \ldots + 2^{n-3} + 2^{n-2} + 2^{n-1}$$

$$+ 2^0 + \ldots + 2^{n-3} + 2^{n-2}$$

$$+ 2^0 + \ldots + 2^{n-3}$$

$$\vdots$$

$$+ 2^0$$

$$= (2^n - 1) + \ldots + (2^1 - 1)$$

$$= 2^{n+1} - 2 - n$$

Primitive Operationen

- Grundlegende Berechnungen, die von einem Algorithmus ausgeführt werden
- Ablesbar aus Pseudocode oder Programmstück
- Überwiegend unabhängig von einer (imperativen) Programmiersprache
- Exakte Definition ist nicht bedeutend
- Beispiele
    - einen Ausdruck auswerten
    - einer Variablen einen Wert zuweisen
    - Indexierung in einem Array
    - Aufrufen einer Methode
    - Verlassen einer Methode

Zählen von primitiven Operationen

Durch Untersuchen des Pseudocode können wir die maximale Zahl von primitiven Operationen, die durch einen Algorithmus ausgeführt wurden, als eine Funktion der Eingabegröße bestimmen.

```
Algorithmus arrayMax(A,n)                    # Operationen
  currentMax ← A[0]                          2
  for i ← 1 to n-1 do                        2(n-1)
      if A[i] > currentMax then              2(n-1)
          currentMax ← A[i]                  2(n-1)
  { erhöhe Zähler i }
  return currentMax                          1

                                   Total     6n-3
```

Laufzeit abschätzen

- Der Algorithmus *arrayMax* führt im worst case 6*n* - 3 primitive Operationen aus
- Definiere
  - *a* Zeit, die die schnellste primitive Operation verbraucht hat
  - *b* Zeit, die die langsamste primitive Operation verbraucht hat
- T(n) sei die tatsächliche worst-case Laufzeit von *arrayMax* . Dann ist :

$$a(6n-3) \leq T(n) \leq b(6n-3)$$

- Daher ist die Laufzeit T(n) durch zwei lineare Funktionen beschränkt.

Zuwachsrate der Laufzeit

- Verändern der Hard- und Softwareumgebung
  - beeinflusst T(n) um einen konstanten Faktor, aber
  - ändert die Wachstumsordnung von T(n) nicht
- Das lineare Wachstum der Laufzeit T(n) ist eine für den Algorithmus *arrayMax* charakteristische Eigenschaft.